# 13 User authentication with OAuth 2.0

***This chapter covers***

- The reasons OAuth 2.0 is *not* an authentication protocol
- Building an authentication protocol using OAuth 2.0
- Identifying and avoiding common mistakes when using OAuth 2.0 in authentication
- Implementing OpenID Connect on top of OAuth 2.0

The OAuth 2.0 specification defines a *delegation* protocol useful for conveying *authorization decisions* across a network of web-enabled applications and APIs. Because OAuth 2.0 is used to gather the consent of an authenticated end user, many developers and API providers have concluded that OAuth 2.0 is an *authentication* protocol that can be used to log in users securely. However, in spite of it being a security protocol that makes use of user interaction, OAuth 2.0 is not an authentication protocol. Let's say that again, to be clear:

***OAuth 2.0 is not an authentication protocol.***

Much of the confusion comes from the fact that OAuth 2.0 is commonly used *inside* of authentication protocols, and that OAuth 2.0 embeds several authentication events inside of a regular OAuth 2.0 process. As a consequence, many developers will see the OAuth 2.0 process and assume that by using OAuth, they're performing user authentication. This turns out to be not only untrue but also dangerous for service providers, developers, and end users.

## 13.1 Why OAuth 2.0 is not an authentication protocol

First we need to answer a fundamental question: What is authentication, anyway? *Authentication,* in this context, is what tells an application who the current user is and whether they're currently using your application. It's the piece of the security architecture that tells you the user is who they claim to be, usually by providing a set of credentials (such as a username and password) to the application that prove this fact. A practical authentication protocol will probably also tell you a number of identity attributes about this user, such as a unique identifier, an email address, and a name to use when the application says, "Good Morning."

However, OAuth 2.0 tells the application none of that. OAuth 2.0, on its own, says absolutely nothing about the user, nor does it say how the user proved their presence, or even if the user is present at all. As far as an OAuth 2.0 client is concerned, it asked for a token, it got a token, and it eventually used that token to access some API. It doesn't know anything about who authorized the application or whether there was even a user there at all. In fact, many of the major use cases for OAuth 2.0 are about obtaining an access token for use when the user isn't able to interactively authorize the application any longer. Thinking back to our printing example, although it's true that the user logged in to both the printing service and the storage service, the user is in no way directly involved in the connection *between* the printing service and the storage service. Instead, the OAuth 2.0 access token allowed the printing service to act on the user's behalf. This is a powerful paradigm for delegated client *authorization*, but it's rather antithetical to *authentication,* in which the whole point is figuring out whether the user is there and who they are.

### 13.1.1 Authentication vs. authorization: a delicious metaphor

To help clear things up, it may be helpful to think of the difference between authentication and authorization in terms of a metaphor: *fudge* and *chocolate*.[1] Although there are some superficial similarities, the nature of these two items is clearly different: chocolate is an ingredient whereas fudge is a confection. You can make chocolate fudge, which is a truly delicious thing in the opinion of your humble authors. This treat is clearly defined by its chocolaty character. As such, it's tempting—but ultimately incorrect—to say that chocolate and fudge are equivalent. Let's unpack that a bit here, and see what on earth it has to do with OAuth 2.0.

*Chocolate* can be used to make many different things in many different forms, but it's always based on cacao. It's a versatile and useful component that lends its distinct flavor to everything from cakes and ice creams to pastry fillings and mole sauce. You can even enjoy chocolate completely on its own with no other ingredients, though even then it can take a number of different forms. One other popular item that can

[1] Much thanks to Vittorio Bertocci for this excellent metaphor, from the blog post "OAuth 2.0 and Sign-In," available at http://www.cloudidentity.com/blog/2013/01/02/oauth-2-0-and-sign-in-4/

be made with chocolate is, of course, chocolate fudge. Here it's clear to the fudge consumer that the chocolate is the star ingredient of this particular confection.

OAuth 2.0, in this metaphor, is chocolate. It's a versatile ingredient fundamental to a number of different security architectures on the web today. OAuth 2.0's delegation model is distinctive, and it's always made up of the same roles and actors. OAuth 2.0 can be used to protect RESTful APIs and web resources. It can be used by clients on web servers and native applications. It can be used by end users to delegate limited authority and by trusted applications to transmit back-channel data. OAuth 2.0 can even be used to make an identity and authentication API, where it's clear that OAuth 2.0 is the key enabling technology.

*Fudge*, conversely, is a confection that can be made out of many different things and it takes on their flavor: from peanut butter to coconut, from oranges to potatoes.[2] In spite of the variety of flavors, fudge always has a particular form and texture that makes it recognizable as fudge, as opposed to some other flavored confection such as mousse or ganache. One popular flavor of fudge is, of course, chocolate fudge. Even though it's clear that chocolate is the star ingredient in this confection, it takes several additional ingredients and a few key processes to transform chocolate into chocolate fudge. The result is something recognizable as chocolate in flavor but fudge in form, and using chocolate to make fudge does not make chocolate equal to fudge.

Authentication in our metaphor is more like fudge. A few key components and processes must be brought together in the right way to make it work properly and securely, and there is a wide variety of options for those components and processes. Users could be required, for example, to carry a device, memorize a secret password, present a biometric sample, prove that they can log in to another remote server, or any number of other approaches. To do their job, these systems can use public key infrastructure (PKI) and certificates, federated trust frameworks, browser cookies, or even proprietary hardware and software. OAuth 2.0 can be one of these technology components, but of course it doesn't have to be. Without other factors, OAuth 2.0 isn't sufficient to carry user authentication.

As there are recipes for making chocolate fudge, there are patterns for making OAuth-based authentication protocols. A number of these are made for specific providers, such as Facebook, Twitter, LinkedIn, or GitHub, and there are even open standards such as OpenID Connect that can work across many different providers. These protocols all start with a common base of OAuth and use their own additional components to provide authentication capabilities in slightly different ways.

## 13.2 Mapping OAuth to an authentication protocol

How, then, can we build an authentication protocol with OAuth as a base? First, we need to map the different OAuth 2.0 parties on to the appropriate parts of an authentication transaction. In an OAuth 2.0 transaction, a resource owner authorizes

[2] No joke, potato fudge is surprisingly good.

a client to access a protected resource using a token from an authorization server. In an authentication transaction, an end user logs in to a relying party (RP) using an identity provider (IdP). With this in mind, a common approach at designing an authentication protocol such as this is to map the relying party on to the protected resource (figure 13.1). After all, isn't the relying party the component *protected* by the authentication protocol?

Although this may seem to be a sensible way to deploy an identity protocol on top of OAuth 2.0, we can see in figure 13.1 that the security boundaries don't line up well. In OAuth 2.0, the client and the resource owner are working together—the client is acting on behalf of the resource owner. The authorization server and protected resource also work together, as the authorization server generates the tokens accepted by the protected resource. In other words, there's a security and trust boundary between the user/client and authorization server/protected resource, and OAuth 2.0 is the protocol used to cross that boundary. When we try to map things, as in figure 13.1, the boundary is now between the IdP and the protected resource. This forces an unnatural crossing of this security boundary, where the protected resource is now interacting directly with the user. However, in OAuth 2.0, the resource owner never generally interacts with the protected resource: it's an API meant to be called by the client application. Remember from the coding exercises in previous chapters, our protected resource doesn't even have a UI to speak of. The client, which does interact with the user, is nowhere to be found in this new mapping.

That doesn't work, and we need to try something else that respects these security boundaries. Let's try to make the RP out of the OAuth 2.0 client, since that's the

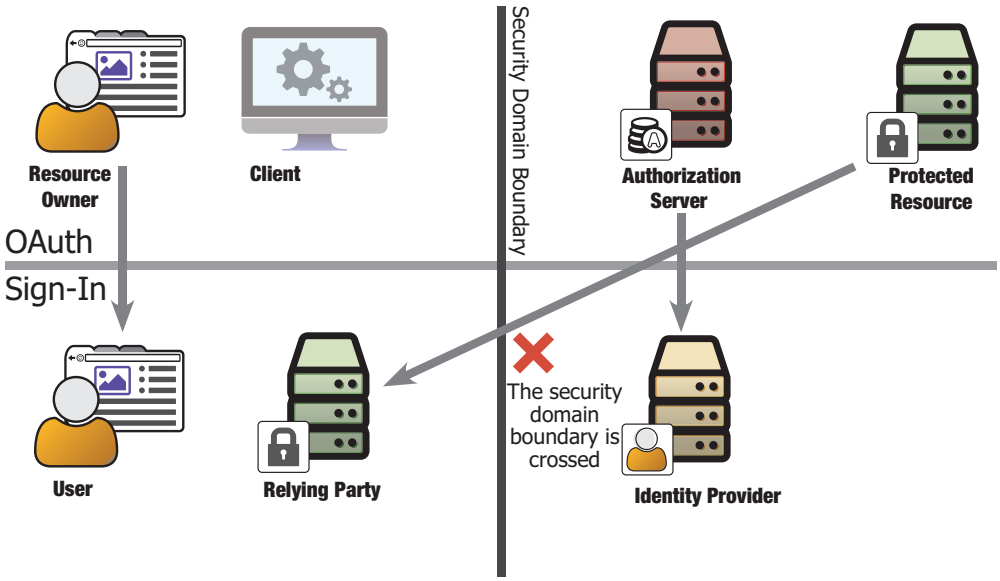


**Figure 13.1 Attempting to make an authentication protocol out of OAuth, and failing**

component that the end user, our resource owner, is normally interacting with anyway. We'll also combine the authorization server and protected resource into a single component, the IdP. We're going to have the resource owner delegate access to the client, but the resource they're delegating access to is their own identity information. That is to say, they're authorizing the RP to find out *who is here right now*, which is of course the essence of the authentication transaction that we're trying to build (figure 13.2).

Although it may seem somewhat counterintuitive to build authentication on top of authorization, we can see here that leveraging the OAuth 2.0 security delegation model gives us a powerful means for connecting systems. Furthermore, notice that we can cleanly map all parts of the OAuth 2.0 system into their corresponding components in an authorization protocol. If we extend OAuth 2.0 so that the information coming from the authorization server and protected resource conveys information about the user and their authentication context, we can give the client everything it needs to log the user in securely.

Now we've got an authentication protocol made up of our familiar OAuth 2.0 pieces. Since we're working in a new protocol space, they get different names. The client is now the relying party, or RP, and the two terms can be used interchangeably for this protocol. We've conceptually combined the authorization server and protected resource into the Identity Provider, or IdP. It's possible that the two aspects of the service, issuing tokens and serving user identity information, could be served by separate servers, but as far as the RP is concerned, they're functioning as a single unit. We're also going to add a second token alongside the access token, and we'll use this new ID token to carry information about the authentication event itself (figure 13.3).

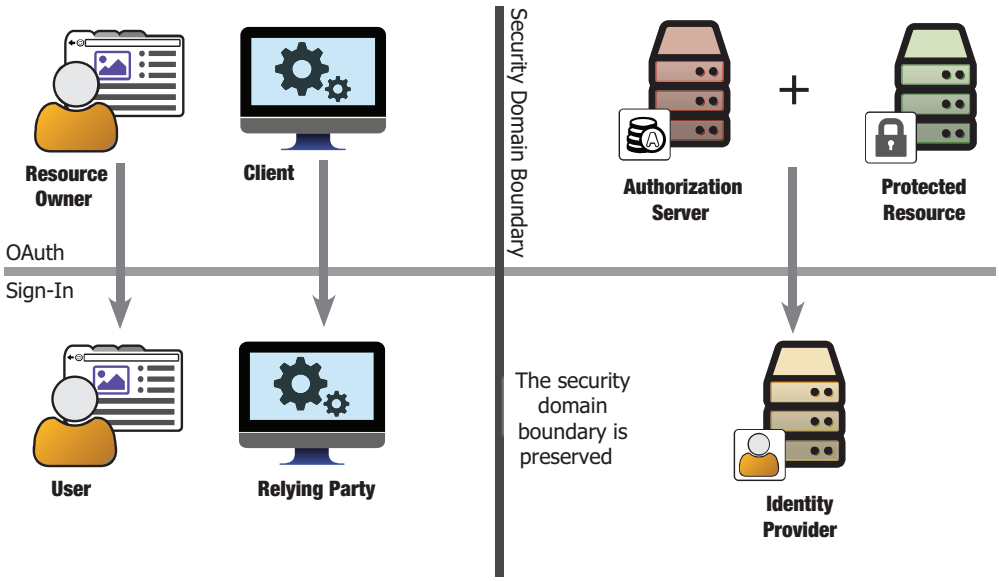


**Figure 13.2 Making an authentication protocol out of OAuth, more successfully**

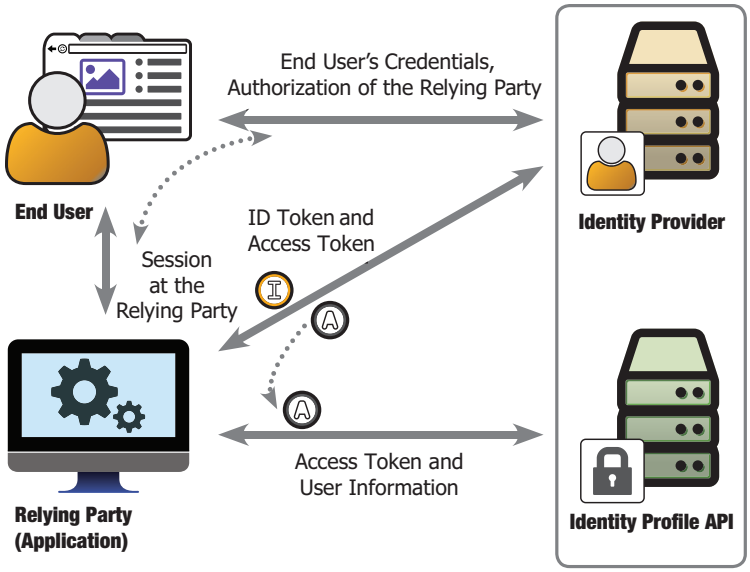


**Figure 13.3 Components of an OAuth-based authentication and identity protocol**

The RP can now find out who the user is and how they logged in, but why are we using two tokens here? We could provide information about the user directly in the token received from the authorization server, or we could provide a user information API that can be called as an OAuth protected resource. As it turns out, there is value in doing both, and we'll look at the way the OpenID Connect protocol does this later in this chapter. To accomplish this, we have two tokens used in parallel with each other, and we'll look at some of the details in a bit.

## 13.3 How OAuth 2.0 uses authentication

In the previous section, we saw how it's possible to build an authentication protocol on top of an authorization protocol. However, it's also true an OAuth transaction requires several forms of authentication to take place in order for the authorization delegation process to function: the resource owner authenticates to the authorization server's authorization endpoint, the client authenticates to the authorization server at the token endpoint, and there may be others depending on the setup. We're building authentication on top of authorization, and the authorization protocol itself relies on authentication, isn't that a bit overcomplicated?

It may seem an odd setup, but notice that this setup can leverage the fact that the user is authenticating at the authorization server, but at no point are the end user's original credentials communicated to the client application (our RP) through the OAuth 2.0 protocol. By limiting the information that each party needs, the transaction can be made much more secure and less prone to failure, and it can function across

security domains. The user authenticates directly to a single party, as does the client, and neither needs to impersonate the other.

Another major benefit of building authentication on top of authorization in this way is that it allows for end-user consent at runtime. By allowing the end user to decide which applications they release their identity to, an OAuth 2.0–based identity protocol can scale across security domains to the internet at large. Instead of organizations needing to decide ahead of time whether all of their users are allowed to log in at another system, each individual user can decide to log in where they choose. This fits the Trust On First Use (TOFU) model of OAuth 2.0 that we first saw in chapter 2.

Additionally, the user can delegate access to other protected APIs alongside their identity. With one call, an application can find out whether a user is logged in, what the application should call the user, download photos for printing, and post updates to the user's message stream. If a service is already offering an OAuth 2.0–protected API, it's not much of a stretch to start offering authentication services as well. This ability to add on services including identity has proved useful in the API-driven world of the web today.

All of this fits nicely within the OAuth 2.0 access model, and this simplicity is compelling. However, by accessing both identity and authorization at the same time, many developers conflate the two functions. Let's take a look at a few common mistakes that this setup can lead to.

## 13.4 Common pitfalls of using OAuth 2.0 for authentication

We've demonstrated that it's possible to build an authentication protocol on top of OAuth, yet there are a number of things that tend to trip up those who do so. These mistakes can happen either on the side of the identity provider or on the side of the identity consumer, and most come from misunderstandings of what different parts of the protocol say.

### 13.4.1 Access tokens as proof of authentication

Since the resource owner usually needs to authenticate at the authorization endpoint prior to an access token being issued, it's tempting to consider reception of an access token as proof of that authentication. However, the token itself conveys no information about the authentication event, or whether an authentication event even occurred during this transaction. After all, the token could have been issued from a long-running (and potentially hijacked) session, or it could have been automatically authorized for some non-personal scope. The token could have been issued directly to the client using an OAuth 2.0 grant type that doesn't require user interaction such as the client credentials, assertion, or refresh token call. Additionally, if the client isn't careful about where it accepts tokens from, the token could have been issued to a different client and injected (see section 13.4.3 for details on this situation).

In any event, no matter how it got the token, the client can't tell anything about the user or their authentication status from the access token. This stems from the fact that the client is not the intended audience of the OAuth 2.0 access token. In OAuth 2.0, the access token is designed to be opaque to the client, but the client needs to be able to derive some user information from the token. Instead, the client is the *presenter* of the access token, and the *audience* is the protected resource.

Now, we could define a token format that the client could parse and understand. This token would carry information about the user and authentication context that the client could read and validate. However, general OAuth 2.0 doesn't define a specific format or structure for the access token, and many existing deployments of OAuth have their own token formats. Furthermore, the life of the access token is likely to outlive the authentication event that would be represented in this token structure. Since the token is passed to protected resources, some of which have nothing to do with identity, it would also be potentially problematic for these protected resources to learn sensitive information about the user's login event. To overcome these limitations, protocols such as OpenID Connect's ID token and Facebook Connect's Signed Response provide a secondary token alongside the access token that communicates the authentication information directly to the client. This allows the primary access token to remain opaque to the client, as in regular OAuth, whereas the secondary authentication token can be well-defined and parsed.

### 13.4.2 Access of a protected API as proof of authentication

Even if the client can't understand the token, it can always present the token to a protected resource that can. What if we define a protected resource that tells the client who issued the token? Since the access token can be traded for a set of user attributes, it's tempting to think that possession of a valid access token is enough to prove that a user is authenticated.

This line of thinking turns out to be true in some cases, but only when the access token was freshly minted in the context of a user being authenticated at the authorization server. Remember, though, this isn't the only way to obtain an access token in OAuth. Refresh tokens and assertions can be used to obtain access tokens without the user being present, and in some cases access grants can occur without the user having to authenticate at all.

Furthermore, the access token will generally be usable long after the user is no longer present. The protected resource isn't generally going to be in a position to tell from the token alone whether the user is present, since by the nature of the OAuth 2.0 protocol the user won't be present on the connection between the client and protected resource. In many larger OAuth 2.0 ecosystems, the user has no means of ever authenticating to the protected resource. Although the protected resource can probably tell which user originally authorized the token, it will generally be hard pressed to say anything about that user's current state.

This becomes especially problematic when there is a large gap of time between the authorization event and the use of the token at the protected resource. OAuth 2.0 can work well when the user is no longer present at either the client or the authorization server, but because the entire point of an authentication protocol is to know whether the user is present, the client can't rely on the presence of a functioning access token to determine whether the user's there. The client can counter this problem by only checking for user information when it knows the token is relatively fresh, and by not assuming a user is present just because the user API can be accessed by a given access token. We could also counter this by having an artifact directed to the client directly that it knows to accept only directly from the IdP, such as the ID token and signed request discussed in the previous section. These tokens have a separate lifecycle from the access tokens, and their contents can be used alongside any additional information from a protected resource.

### 13.4.3 Injection of access tokens

An additional (and dangerous) threat occurs when clients accept access tokens from sources other than the return of an intentional request to token endpoint. This is especially troublesome for a client that uses the implicit flow, in which the token is passed directly to the client as a parameter in the URL hash. An attacker can take an access token, either a valid one from a different application or a spoofed one, and pass it to the waiting RP as if it were requested by that RP. This is problematic enough in plain OAuth 2.0, in which the client can be tricked into accessing resources other than those of the real resource owner, but it is utterly disastrous in an authentication protocol because it would allow an attacker to copy tokens and use them to log in to another application.

This issue can also occur if different parts of an application pass the access token between components in order to "share" access among them. This is problematic because it opens up a place for access tokens to potentially be injected into an application by an outside party and potentially leak outside of the application. If the client application doesn't validate the access token through some mechanism, it has no way of differentiating between a valid token and an attacker's token.

This can be mitigated by using the authorization code flow instead of the implicit flow, which would mean the client would accept tokens only directly from the authorization server's token endpoint. The `state` parameter allows a client to provide a value that is unguessable by an attacker. If this parameter is absent or it doesn't align with an expected value, the client can easily reject the incoming token as invalid.

### 13.4.4 Lack of audience restriction

Most OAuth 2.0 APIs don't provide any mechanism of audience restriction for their returned information. That is to say, there's no way for a client to tell whether an access token was intended for it or another client. It's possible to take a naive client, hand it a (valid) token from another client, and have the naive client call a user API. Since

the protected resource doesn't know the identity of the client making the call, only the validity of the token, this act will return valid user information. However, this information was intended for consumption by another client. The user hasn't even authorized the naive client, and yet it treats the user as logged in.

This problem can be mitigated by communicating the authentication information to a client along with an identifier that the client can recognize and validate as its own. This will allow the client to differentiate between an authentication for itself and an authentication for another application. This attack can be further mitigated by passing the set of authentication information directly to the client during the OAuth 2.0 process instead of through a secondary mechanism such as an OAuth 2.0–protected API, preventing a client from having an unknown and untrusted set of information injected later in the process.

### 13.4.5 Injection of invalid user information

If an attacker is able to intercept or co-opt one of the calls from the client, it could alter the content of the returned user information without the client being able to know anything was amiss. This would allow an attacker to impersonate a user at a naive client by swapping out a user identifier in the right call sequence, for example, in the return value from a user information API or inside a token directed at the client.

This attack can be mitigated by cryptographically protecting and verifying the authentication information as it's passed to the client. All communication pathways between the client and the authorization server need to be protected by TLS, and the client needs to verify the server's certificate when it connects. In addition, the user information or the token (or both) can be signed by the server and verified by the client. This additional signature will prevent an adversary from altering or injecting user information even if they're able to hijack the connection between the parties.

### 13.4.6 Different protocols for every potential identity provider

One of the biggest problems with OAuth 2.0–based identity APIs is that different identity providers will inevitably implement the details of the identity API differently, even if they're using fully standards-compliant OAuth as the basis. For example, a user's unique identifier might be found in a `user_id` field in one provider but in the `sub` field in another provider. Even though these fields are semantically equivalent, they would require two separate code paths to process. Although the authorization may happen the same way at each provider, the conveyance of the authentication information could be different.

This problem occurs because the mechanisms for conveying authentication information discussed here are explicitly left out of scope for OAuth 2.0. OAuth 2.0 defines no specific token format, defines no common set of scopes for the access token, and doesn't address how a protected resource validates an access token. Consequently, this problem can be mitigated by providers using a standard authentication protocol built

on top of the OAuth standard so that, no matter where the identity information is coming from, it's transmitted in the same way. Is there such a standard?

## 13.5 OpenID Connect: a standard for authentication and identity on top of OAuth 2.0

OpenID Connect[3] is an open standard published[4] by the OpenID Foundation in February 2014 that defines an interoperable way to use OAuth 2.0 to perform user authentication. In essence, it's a widely published "recipe for chocolate fudge" that has been built and tested by a wide variety of implementers. As an open standard, OpenID Connect can be implemented without license or intellectual property concerns. Since the protocol is designed to be interoperable, an OpenID client application can speak one protocol to many identity providers instead of implementing a slightly different protocol to each identity provider.

OpenID Connect is built directly on OAuth 2.0 and remains compatible with it. In many instances, it's deployed along with a plain OAuth infrastructure that protects other APIs. In addition to OAuth 2.0, OpenID Connect uses the JSON Object Signing and Encryption (JOSE) suite of specifications (which we covered in chapter 11) for carrying signed and encrypted information around in different places. An OAuth 2.0 deployment with JOSE capabilities is already far along on the way to being a fully compliant OpenID Connect system, as the delta between the two is relatively small. But that delta makes a big difference, and OpenID Connect manages to avoid many of the pitfalls discussed previously by adding several key components to the OAuth 2.0 base.

### 13.5.1 ID tokens

The OpenID Connect ID token is a signed JSON Web Token (JWT) given to the client application alongside the regular OAuth access token. Unlike the access token, the ID token is directed to the RP and is intended to be parsed by it.

As with the signed access tokens that we created in chapter 11, the ID token contains a set of claims about the authentication session, including an identifier for the user (`sub`), the identifier for the identity provider that issued the token (`iss`), and the identifier of the client for which this token was created (`aud`). Additionally, the ID token contains information about the token's own validity time window (with the `exp` and `iat` claims) as well as any additional information about the authentication context to be conveyed to the client. For example, the token can say how long ago the user was presented with a primary authentication mechanism (`auth_time`) or what kind of primary authentication they used at the IdP (`acr`). The ID token can also have other claims inside it, both standard JWT claims such as those listed in chapter 11 as well as extended claims for the OpenID Connect protocol. The required claims are in boldface in table 13.1.

[3] http://openid.net/connect/

[4] http://openid.net/specs/openid-connect-core-1_0.html

**Table 13.1 Claims inside an ID token**

| Claim Name | Claim Description |
|---|---|
| **iss** | The *issuer* of the token; URL of the IdP. |
| **sub** | The *subject* of the token, a stable and unique identifier for the user at the IdP. This is usually a machine-readable string and shouldn't be used as a username. |
| **aud** | The *audience* of the token; must contain the client ID of the RP. |
| **exp** | The *expiration* timestamp of the token. All ID tokens expire, and usually pretty quickly. |
| **iat** | The timestamp of *when the token was issued*. |
| auth_time | The timestamp of *when the user authenticated to the IdP*. |
| nonce | A string sent by the RP during the authentication request, used to mitigate replay attacks similar to the *state* parameter. It must be included if the RP sends it. |
| acr | The *authentication context reference*, which indicates an overall categorization of the authentication that the user performed at the IdP. |
| amr | The *authentication method reference*, which indicates how the user authenticated to the IdP. |
| azp | The *authorized party* for this token; must contain the client ID of the RP if it's included. |
| at_hash | Cryptographic hash of the access token. |
| c_hash | Cryptographic hash of the authorization code. |

The ID token is issued in addition to an access token as the id_token member of the token endpoint response, not in lieu of it. This is in recognition of the fact that the two tokens have different intended audiences and uses. The two-token approach allows the access token to remain opaque to the client as in regular OAuth while allowing the ID token be parsed. Furthermore, the two tokens can also have different lifecycles, with the ID token often expiring more quickly. Although the ID token represents a single authentication event, and it's never passed to an external service, the access token can be used to fetch protected resources long after the user has left. Although it's true that you could still use the access token to ask who authorized the client in the first place, doing so wouldn't tell you anything about the user's presence, as you saw previously.

```
{
   "access_token": "987tghjkiu6trfghjuytrghj",
   "token_type": "Bearer",
   "id_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwOi8vbG9jY
   Wxob3N0OjkwMDEvIiwic3ViIjoiOVhFMy1KSTM0LTAwMTMyQSIsImF1ZCI6Im9hdXRoLWNsaWVud
   C0xIiwiZXhwIjoxNDQwOTg3NTYxLCJpYXQiOjE0NDA5ODY1NjF9.LC5XJDhxhA5BLcT3VdhyxmMf6
   EmlFM_TpgL4qycbHy7JYsO6j1pGUBmAiXTO4whK1qlUdjR5kUm ICcYa5foJUfdT9xFGDtQhRcG3-
   dOg2oxhX2r7nhCjzUnOIebr5POySGQ8ljT0cLm45edv_rO5fSVPdwYGSa7QGdhB0bJ8KJ__Rs
   yKB707n09y1d92ALwAfaQVoyCjYB0uiZM9Jb8yHsvyMEudvSD5urRuHnGny8YlGDIofP6SXh5-
   1TlR7ST7R7h9f4Pa0lD9SXEzGUG816HjIFOcD4aAJXxn_QMlRGSfL8NlIz29PrZ2xqg8w2w84hBQ
   cgchAmj1TvaT8ogg6w"
}
```

Finally, the ID token itself is signed by the identity provider's key, adding another layer of protection to the claims inside it in addition to the TLS transport protection that was used to get the token in the first place. Since the ID token is signed by the authorization server, it also provides a location to add detached signatures over the authorization code (`c_hash`) and access token (`at_hash`). The client can validate these hashes while still keeping the authorization code and access token content opaque to the client, preventing a whole class of injection attacks.

By applying a few simple checks to this ID token, the same checks used when processing a signed JWT as we did in chapter 11, a client can protect itself from a large number of common attacks:

1. Parse the ID token to ensure it's a valid JWT and collect the claims.
   - Split the string on the "." character.
   - Base64URL decode each section.
   - Parse the first two sections (header and payload) as JSON.
2. Validate the signature of the token against the public key of the IdP, published at a discoverable location.
3. See that the ID token is issued by a trusted IdP.
4. Ensure the client's own client identifier is included in the audience list of the ID token.
5. Confirm that the expiration, issued-at, and not-before timestamp values are reasonable given the current time.
6. Make sure that the `nonce`, if present, matches the one sent out.
7. Validate the hashes for the authorization code or access token, if applicable.

Each of these steps is deterministic and mechanical, requiring minimal coding effort. Some more advanced modes of OpenID Connect allow for the ID token to be encrypted as well, which changes the parsing and verification process slightly but with the same results.

### 13.5.2 The UserInfo endpoint

Since the ID token contains all the necessary information for processing the authentication event, OpenID Connect clients don't need anything more than this to process a successful login. However, the access token can be used at a standard protected resource that contains profile information about the current user, which is called the UserInfo endpoint. The claims at this endpoint aren't part of the authentication process previously discussed but instead provide bundled identity attributes that make the authentication protocol more valuable to application developers. After all, it's preferable to say, "Good Morning, Alice" instead of, "Good Morning, 9XE3-JI34-00132A."

The request to the UserInfo endpoint is a simple HTTP GET or POST, with the access token (*not the ID token*) sent as the authorization. There are no input parameters in a normal request, though like with much of OpenID Connect there are some

advanced methods that can be used here. The UserInfo endpoint follows a protected resource design of having the same resource for all users in the system, as opposed to a different resource URI for each user. The IdP figures out which user is being asked about by dereferencing the access token.

```
GET /userinfo HTTP/1.1
Host: localhost:9002
Accept: application/json
```

The response from the UserInfo endpoint is a JSON object that contains claims about the user. These claims tend to be stable over time, and it's common to cache the results of the UserInfo endpoint call instead of fetching them on every authentication request. Using advanced capabilities of OpenID Connect, it's also possible to return the UserInfo response as a signed or encrypted JWT.

```
HTTP/1.1 200 OK
Content-type: application/json

{
   "sub": "9XE3-JI34-00132A",
   "preferred_username": "alice",
   "name": "Alice",
   "email": "alice.wonderland@example.com",
   "email_verified": true
}
```

OpenID Connect uses the special `openid` scope value to gate access to the UserInfo endpoint. OpenID Connect defines a set of standardized OAuth scopes that map to subsets of these user attributes (`profile`, `email`, `phone`, and `address`, shown in table 13.2), allowing plain OAuth transactions to request everything necessary for an authentication. The OpenID Connect specification goes into greater detail for each of these scopes and what attributes they map to.

OpenID Connect defines a special `openid` scope that controls overall access to the UserInfo endpoint by the access token. The OpenID Connect scopes can be used alongside other non–OpenID Connect OAuth 2.0 scopes without conflict, and the

**Table 13.2 Mapping OAuth scopes to OpenID Connect UserInfo claims**

| Scope | Claims |
|---|---|
| **openid** | **sub** |
| profile | name, family_name, given_name, middle_name, nickname, preferred_username, profile, picture, website, gender, birthdate, zoneinfo, locale, updated_at |
| email | email, email_verified |
| address | address, a JSON object which itself contains formatted, street_address, locality, region, postal_code, country |
| phone | phone_number, phone_number_verified |

access token issued can potentially be targeted at several different protected resources in addition to the UserInfo endpoint. This approach allows an OpenID Connect identity system to smoothly coexist with an OAuth 2.0 authorization system.

### 13.5.3 Dynamic server discovery and client registration

OAuth 2.0 was written to allow a variety of different deployments, but by design doesn't specify how these deployments come to be set up or how the components know about each other. This is acceptable in the regular OAuth world in which one authorization server protects a specific API, and the two are usually closely coupled. OpenID Connect defines a common API that can be deployed across a wide variety of clients and providers. It would not be scalable for each client to have to know ahead of time about each provider, nor would it be at all reasonable to require each provider to know about every potential client.

To counteract this, OpenID Connect defines a discovery protocol[5] that allows clients to easily fetch information on how to interact with a specific identity provider. This discovery process happens in two steps. First, the client needs to discover the issuer URL of the IdP. This can be configured directly, such as in a common NASCAR-style provider chooser in figure 13.4.

Alternatively, the issuer can be discovered based on the WebFinger protocol. WebFinger works by taking a common means of user identification, email addresses, and provides a set of deterministic transformation rules that takes this friendly user-facing input and outputs a discovery URI (figure 13.5). In essence, you take the domain portion of the email address identifier, append `https://` to the front, and append `/.well-know/webfinger` to the end of it to create a URI. Optionally, you can also pass in information about what the user originally typed in as well as the kind of information you're looking for. In OpenID Connect, this discovery URI can be fetched over HTTPS to determine the issuer for a particular user's address.


**Figure 13.4 A NASCAR-style identity provider selector**

[5] http://openid.net/specs/openid-connect-discovery-1_0.html

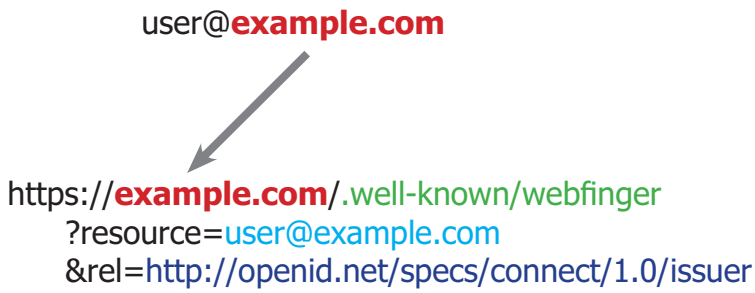


**Figure 13.5 WebFinger transforms an email address into a URL**

After the issuer is determined, the client still needs essential information about the server, such as the location of the authorization and token endpoints. This is discovered by appending `/.well-known/openid-configuration` to the issuer URI discovered in the first step and fetching the resulting URL. This returns a JSON document containing all of the attributes of the server that the client needs in order to start the authentication transaction. The following is an example adapted from a publicly available test server:

```
{
"issuer": "https://example.com/",
"request_parameter_supported": true,
"registration_endpoint": "https://example.com/register",
"token_endpoint": "https://example.com/token",
"token_endpoint_auth_methods_supported":
[ "client_secret_post", "client_secret_basic", "client_secret_jwt",
   "private_key_jwt", "none" ],
"jwks_uri": "https://example.com/jwk",
"id_token_signing_alg_values_supported":
[ "HS256", "HS384", "HS512", "RS256", "RS384", "RS512", "ES256", "ES384",
   "ES512", "PS256", "PS384", "PS512", "none" ],
"authorization_endpoint": "https://example.com/authorize",
"introspection_endpoint": "https://example.com/introspect",
"service_documentation": "https://example.com/about",
"response_types_supported":
[ "code", "token" ],
"token_endpoint_auth_signing_alg_values_supported":
[ "HS256", "HS384", "HS512", "RS256", "RS384", "RS512", "ES256", "ES384",
   "ES512", "PS256", "PS384", "PS512" ],
"revocation_endpoint": "https://example.com/revoke",
"grant_types_supported":
[ "authorization_code", "implicit", "urn:ietf:params:oauth:grant-
   type:jwt-bearer", "client_credentials", "urn:ietf:params:oauth:grant_
   type:redelegate" ],
"scopes_supported":
[ "profile", "email", "address", "phone", "offline_access", "openid" ],
"userinfo_endpoint": "https://example.com/userinfo",
"op_tos_uri": "https://example.com/about",
"op_policy_uri": "https://example.com/about",
}
```

Once the client knows about the server, the server needs to know about the client. For this, OpenID Connect defines a client registration protocol[6] that allows clients to be introduced to new identity providers. The OAuth Dynamic Client Registration protocol extension, discussed in chapter 12, was developed in parallel to the OpenID Connect version, and the two are compatible with each other on the wire.

By leveraging discovery, registration, a common identity API, and end-user choice, OpenID Connect can function at internet scale. Even when no parties have to know about each other ahead of time, two compliant OpenID Connect instances can interact with each other to effect an authorization protocol across security boundaries.

### 13.5.4 Compatibility with OAuth 2.0

Even with all of this robust authentication capability, OpenID Connect is by design still compatible with plain OAuth 2.0. In fact, if a service is already using OAuth 2.0 and the JOSE specifications, including JWT, that service is already well on its way to supporting OpenID Connect in full.

To facilitate the building of good client applications, the OpenID Connect working group has published documents on building a basic OpenID Connect client[7] using the authorization code flow as well as building an implicit OpenID Connect client.[8] Both of these documents walk the developer through building a basic OAuth 2.0 client and adding the handful of components necessary for OpenID Connect functionality, many of which have been described here.

### 13.5.5 Advanced capabilities

Although the core of the OpenID Connect specification is fairly straightforward, not all use cases can be adequately addressed by the base mechanisms. To support many advanced use cases, OpenID Connect also defines a number of optional advanced capabilities beyond standard OAuth. Covering all of these in depth could easily fill another book,[9] but we can at least touch on a few key components in this section.

An OpenID Connect client can optionally *authenticate using a signed JWT* in lieu of OAuth's more traditional shared client secret. This JWT can be signed with a client's asymmetric key if it registers its public key with the server, or it can be signed symmetrically with the client secret. This method provides a higher level of security for clients, which avoid sending their passwords across the network.

Similarly, an OpenID Connect client can optionally send its *requests to the authorization endpoint as a signed JWT* instead of a set of form parameters. As long as the key used to sign this request object is registered with the server, the server can validate the parameters inside the request object and be assured that the browser did not tamper with them.

---

[6] http://openid.net/specs/openid-connect-registration-1_0.html
[7] http://openid.net/specs/openid-connect-basic-1_0.html
[8] http://openid.net/specs/openid-connect-implicit-1_0.html
[9] If you think that's a good idea, please contact our publisher and let them know!

An OpenID Connect server can optionally *sign or encrypt the output from the server*, including the UserInfo endpoint, as a JWT. The ID token can likewise be encrypted in addition to being signed by the server. These protections can assure the client that the output was not tampered with, in addition to the assurances garnered from using TLS on the connection.

Other parameters have been added as extensions to the OAuth 2.0 endpoints, including hints for *display types, prompting behavior, and authentication context references*. Using the request object construct, an OpenID Connect client can make much more fine-tuned requests of the authorization server than its OAuth 2.0 counterparts thanks to the inherent expressivity of the request object's JSON payload. These requests can include fine-grained user claims information, such as requesting that only a user matching a specific identifier be logged in.

OpenID Connect provides a way for the *server (or another third party) to initiate the login process*. Although all canonical OAuth 2.0 transactions are initiated by the client application, this optional feature gives the client a way to receive signals to start the login process with a specific IdP.

OpenID Connect also defines a few different ways to retrieve tokens, including *hybrid flows* whereby some information (such as the ID token) is conveyed on the front channel and other information (such as the access token) is conveyed on the back channel. These flows should not be thought of as simple combinations of existing OAuth 2.0 flows, but instead as new functionality for different applications.

Finally, OpenID Connect provides a specification for *managing sessions* between the RP and IdP, or even between multiple RPs. Since OAuth 2.0 has no notion of the user being present apart from the moment of authorization delegation, extensions are required for handling the lifecycle of a federated authentication. If the user logs out from one RP, they may want to log out from others as well, and the RP needs to be able to signal the IdP that this should happen. Other RPs need to be able to listen for a signal from the IdP that a logout has taken place and be able to react accordingly.

OpenID Connect provides all of these extensions without breaking compatibility with OAuth 2.0.

## 13.6 Building a simple OpenID Connect system

Open up `ch-13-ex-1` to find a fully functional OAuth 2.0 system. We're now going to build a simple OpenID Connect system on top of our existing OAuth 2.0 infrastructure. Although an entire book could be dedicated to implementing all of the features of OpenID Connect, we are going to cover the basics here in this exercise. We'll be adding support for issuing the ID token to the authorization code flow on our authorization server. We'll also be building a UserInfo endpoint into our protected resource with a shared database, because this is a common deployment pattern. Notice that even though our authorization server and UserInfo endpoint are running in separate processes, from the RP's perspective they're functioning as a single IdP. We'll also be making our generic OAuth 2.0 client into an OpenID Connect RP by parsing and validating the ID token and fetching the UserInfo for display.

In all of these exercises, we've left out one key component: authenticating the user. Instead, we're once again using a simple drop-down selection on the authorization page to determine which user is "logged in" to the IdP, as we did in chapter 3. In a production system, the primary authentication mechanism used at the IdP is of utmost importance, as the federated identity issued by the server hinges on this. Many good primary authentication libraries exist, and incorporating them to our framework is left as an exercise to the reader. But still, in case it needs to be said: please don't use a simple drop-down box as the authentication mechanism in your production system.

### 13.6.1 Generating the ID token

First, we need to generate an ID token and hand it out alongside our access token. We'll use the same libraries and techniques that we used in chapter 11, since an ID token is really just a special JWT. If you want details on JWTs, head back to chapter 11 for more.

Open up `authorizationServer.js` in an editor. Up near the top of the file, we've supplied user information for two users in the system, Alice and Bob. We'll need this for creating both the ID token and the UserInfo response. For simplicity, we've opted for a simple in-memory variable indexed by the username selectable from the drop-down menu on the authorization page. In a production system, this would likely be tied into a database, directory service, or other persistent store.

```
var userInfo = {

   "alice": {
        "sub": "9XE3-JI34-00132A",
        "preferred_username": "alice",
        "name": "Alice",
        "email": "alice.wonderland@example.com",
        "email_verified": true
   },

   "bob": {
        "sub": "1ZT5-OE63-57383B",
        "preferred_username": "bob",
        "name": "Bob",
        "email": "bob.loblob@example.net",
        "email_verified": false
   }

};
```

Next, we'll create the ID token after we've already created our access token. First we need to determine whether or not we're supposed to be creating an ID token at all. We want to generate an ID token only if the user authorized the `openid` scope, and if we've got a user to speak of at all.

```
if (__.contains(code.scope, 'openid') && code.user) {
```

Next we'll create a header for our ID token and add all of the fields required for the payload. First we set our authorization server as the issuer and add the subject

identifier of the user. Remember, these two fields together give a globally unique identifier for the user. We'll then set the client ID of the requesting client to the audience of the token. Finally, we'll timestamp the token and set an expiration of five minutes in the future. This is generally more than enough time for an ID token to be processed and tied to a user session at an RP. Remember that the RP doesn't have to use the ID token at any external resource, so the timeout can and should be relatively short.

```
var header = { 'typ': 'JWT', 'alg': rsaKey.alg, 'kid': rsaKey.kid };

var ipayload = {
   iss: 'http://localhost:9001/',
   sub: code.user.sub,
   aud: client.client_id,
   iat: Math.floor(Date.now() / 1000),
   exp: Math.floor(Date.now() / 1000) + (5 * 60)
};
```

We'll also add in the `nonce` value, but only if the client sent it on the original request to the authorization endpoint. This value is analogous to the `state` parameter in many ways, but closes a slightly different cross-site attack vector.

```
if (code.request.nonce) {
   ipayload.nonce = code.request.nonce;
}
```

Then we'll sign it with the server's key and serialize it as a JWT.

```
var privateKey = jose.KEYUTIL.getKey(rsaKey);
var id_token = jose.jws.JWS.sign(header.alg, JSON.stringify(header),
   JSON.stringify(ipayload), privateKey);
```

Finally, we'll issue it alongside the access token by modifying the existing token response.

```
token_response.id_token = id_token;
```

And that's all we have to do. Although we could store our ID token along with our other tokens if we wanted to, it's never passed back to the authorization server or any protected resource; therefore, there's no real need to do so. Instead of acting like an access token, it acts like an assertion from the authorization server to the client. Once we send it to the client, we're pretty much done with it.

### 13.6.2 Creating the UserInfo endpoint

Next we'll be adding in the UserInfo endpoint to our protected resource. Open up `protectedResource.js` for this part of the exercise. Notice that while the IdP is a single logical component in the OpenID protocol, it's acceptable and valid to implement it as separate servers are we're doing here. We've imported the `getAccessToken` and `requireAccessToken` helper functions from previous exercises. These will use the local database to look up not only the token information but also the user information associated with the token. Our IdP will be serving user information from `/userinfo` in response to HTTP GET or POST requests. Due to limitations in the

Express.js framework that we're using in our code, we have to define this slightly differently from previous exercises by using an externally named function variable for our handler code, but the effect is roughly the same as before.

```
var userInfoEndpoint = function(req, res) {

};

app.get('/userinfo', getAccessToken, requireAccessToken, userInfoEndpoint);
app.post('/userinfo', getAccessToken, requireAccessToken, userInfoEndpoint);
```

Next we'll check to make sure that the incoming token contains at least the `openid` scope. If it doesn't, we'll return an error.

```
if (!__.contains(req.access_token.scope, 'openid')) {
   res.status(403).end();
   return;
}
```

Once again, we need to get the right set of user information from our data store. We'll base this on the user that authorized the access token, similarly to how we dispatched information in one of the exercises in chapter 4. If we can't find a user, we'll return an error.

```
var user = req.access_token.user;
if (!user) {
   res.status(404).end();
   return;
}
```

Next we need to build up the response. We can't return the entire user info object, since the user may have authorized only a subset of the available scopes. Because each scope maps to a subset of the user's information, we'll go through each of the scopes in the access token and add the associated claims to our output object as we go.

```
var out = {};
__.each(req.access_token.scope, function (scope) {
   if (scope == 'openid') {
        __.each(['sub'], function(claim) {
                if (user[claim]) {
                        out[claim] = user[claim];
                }
        });
   } else if (scope == 'profile') {
        __.each(['name', 'family_name', 'given_name', 'middle_name',
   'nickname', 'preferred_username', 'profile', 'picture', 'website',
   'gender', 'birthdate', 'zoneinfo', 'locale', 'updated_at'],
   function(claim) {
                if (user[claim]) {
                        out[claim] = user[claim];
                }
        });
   } else if (scope == 'email') {
        __.each(['email', 'email_verified'], function(claim) {
                if (user[claim]) {
```

```
                        out[claim] = user[claim];
                }
        });
    } else if (scope == 'address') {
        __.each(['address'], function(claim) {
                if (user[claim]) {
                        out[claim] = user[claim];
                }
        });
    } else if (scope == 'phone') {
        __.each(['phone_number', 'phone_number_verified'], function(claim) {
                if (user[claim]) {
                        out[claim] = user[claim];
                }
        });
    }
});
```

The end result is an object that contains all of the claims for the correct user that were authorized by that user for this client. This process provides an incredible amount of flexibility in terms of privacy, security, and user choice. We'll return this object as JSON.

```
res.status(200).json(out);
return;
```

The final function looks like listing 14 in appendix B.

With two small additions, we've made our functional OAuth 2.0 server into an OpenID Connect IdP as well. We were able to re-use many of the components that we've explored in previous chapters such as JWT generation (chapter 11), inbound access token processing (chapter 4), and scanning for scopes (chapter 4). There are many additional features to OpenID Connect that we talked about earlier, including request objects, discovery, and registration, but implementation of these is left as an exercise to the reader (or the reader of another book).

### 13.6.3 Parsing the ID token

Now that the server is able to generate ID tokens, the client needs to be able to parse them. We're going to use a similar method to that used in chapter 11 where we parsed and validated a JWT at our protected resource. This time, the token is targeted to the client, so we'll be inside `client.js` in an editor to get started. We've statically configured the client and the server with each other's information, but in OpenID Connect all of this can be done dynamically using dynamic client registration and server discovery. As an added exercise, pull in the dynamic client registration code from chapter 12 and implement server discovery on top of this framework.

First, we need to pull the token value off the token response. Since it's passed to us in the same structure that the access token is in, we'll pull it off that object in our token response parsing function. We'll also throw out any old user information or ID tokens we might have had sitting around from a previous login.

```
if (body.id_token) {
```

```
   userInfo = null;
   id_token = null;
```

After that, we'll parse the ID token's payload into a JSON object and test the content of the ID token, starting with its signature. In OpenID Connect, the client will commonly fetch the server's keys from a JSON Web Key (JWK) set URL, but we've provided it statically in the code alongside the server's configuration. For an added exercise, configure the server to publish its public key and configure the client to fetch the server's key when needed at runtime. Our server uses the RS256 signature method for its ID tokens, and we're using the `jsrsasign` library to handle our JOSE functions, as we did in chapter 11.

```
var pubKey = jose.KEYUTIL.getKey(rsaKey);
var tokenParts = body.id_token.split('.');
var payload = JSON.parse(base64url.decode(tokenParts[1]));
if (jose.jws.JWS.verify(body.id_token, pubKey, [rsaKey.alg])) {
```

Then we need to check a few of the fields to make sure they make sense. Once again, we've pulled out each check into its own nested if statement, only accepting the token if all checks pass. First we'll make sure the issuer matches that of our authorization server, and also that our client ID is in the audience list.

```
if (payload.iss == 'http://localhost:9001/') {
   if ((Array.isArray(payload.aud) && __.contains(payload.aud,
   client.client_id)) ||
        payload.aud == client.client_id) {
```

Then we'll make sure that the issued at and expiration timestamps make sense.

```
var now = Math.floor(Date.now() / 1000);
if (payload.iat <= now) {
   if (payload.exp >= now) {
```

A few extra tests use more advanced forms of the protocol, such as comparing a `nonce` value if we had sent one in the original request or calculating the hashes for the access token or code. These tests aren't needed for a simple client using the authorization code grant type, and they're left as exercises for the reader.

If and only if all of these checks pass, we have a valid ID token that we can save in our application. In fact, we don't need to save the entire token anymore, since we've validated it already, so we're going to save the payload portion so that we can access it later:

```
id_token = payload;
```

Throughout our application, we can use a pairing of the `id_token.iss` and `id_token.sub` values from the ID token as a globally unique identifier for the current user. This technique is much more collision resistant than a username or email address would be because the issuer URL automatically scopes the values in the subject field. Once we've got the ID token, we send the user to an alternate display page showing that they've successfully logged in as the current user.

```
res.render('userinfo', {userInfo: userInfo, id_token: id_token});
return;
```

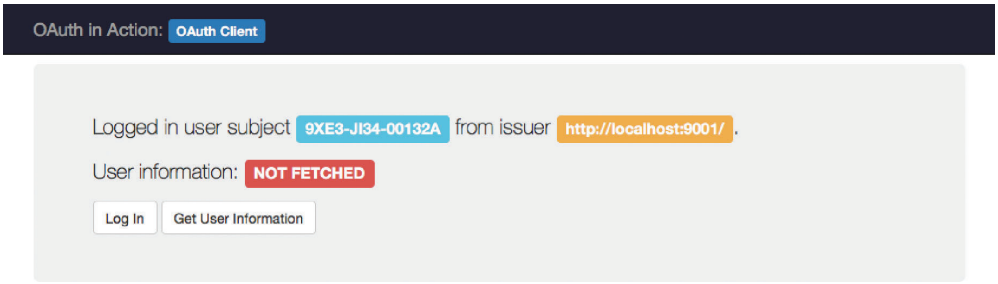


**Figure 13.6 The client page showing a logged-in user**

This gives us a display including the issuer and subject, as well as a button to fetch the UserInfo for the current user. The final processing function looks like listing 15 in appendix B.

### 13.6.4 Fetching the UserInfo

After we've processed the authentication event, chances are we're going to want to know more about the user than a single machine-readable identifier. To access their profile information, including things such as their name and email address, we're going to call the UserInfo endpoint at the IdP using the access token that we received during the OAuth 2.0 process. It's possible that this access token could be used for additional resources as well, but we're going to focus specifically on its use with the UserInfo endpoint.

Instead of automatically downloading the user information immediately upon authentication, we're going to make our RP call the UserInfo endpoint only when needed. In our application, we'll be saving this to the `userInfo` object and rendering it to a web page.

We've already included the rendering template in the project for you, so we'll start by creating a handler function for `/userinfo` on the client.

```
app.get('/userinfo', function(req, res) {

});
```

This call works like any other OAuth 2.0 protected resource. In this specific case, we're going to make an HTTP GET request with the access token in the authorization header.

```
var headers = {
   'Authorization': 'Bearer ' + access_token
};

var resource = request('GET', authServer.userInfoEndpoint,
   {headers: headers}
);
```

The UserInfo endpoint returns a JSON object that we can then save and process how we see fit. If we receive a successful response, we're going to save the user information and hand it to our rendering template. Otherwise, we'll display an error page.

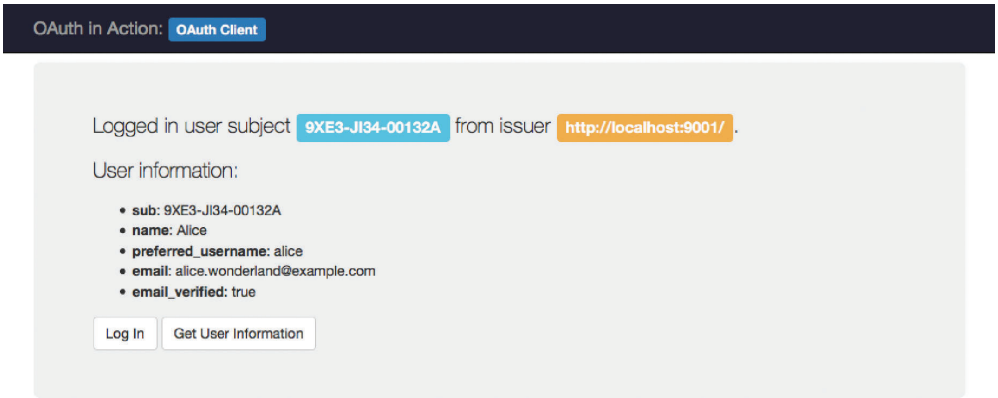


**Figure 13.7 The client showing a successful login and fetch of user information**

```
if (resource.statusCode >= 200 && resource.statusCode < 300) {
   var body = JSON.parse(resource.getBody());

   userInfo = body;

   res.render('userinfo', {userInfo: userInfo, id_token: id_token});
   return;
} else {
   res.render('error', {error: 'Unable to fetch user information'});
   return;
}
```

This should give you a page that looks something like what is shown in figure 13.7. And that's all there is to it. Try authorizing different scopes and looking at the difference it makes in the data that comes back from the endpoint. If you've written an OAuth 2.0 client in the past (which you have, back in chapter 3), then this should all seem trivial, and for good reason: OpenID Connect is designed from the start to be something built on top of OAuth 2.0.

For an added exercise, wire the client's `/userinfo` page to require a valid OpenID Connect login. That is to say, there must be a valid ID token as well as an access token that can be used to fetch user information already stored at the client when someone goes to that page, and if there is not, the client will automatically start the authentication protocol process.

## 13.7 Summary

Many people erroneously believe that OAuth 2.0 is an authentication protocol, but now you know the truth of the matter.

- OAuth 2.0 is not an authentication protocol, but it can be used to build an authentication protocol.
- Many existing authentication protocols that have been built using OAuth 2.0 are in use on the web today, most of them tied to specific providers.

- Designers of authentication protocols make many common mistakes on top of OAuth 2.0. These mistakes can be avoided with careful design of the authentication protocol.
- With a few key additions, the OAuth 2.0 authorization server and protected resource can act as an identity provider, and the OAuth 2.0 client can act as a relying party.
- OpenID Connect provides a carefully designed open standard authentication protocol built on top of OAuth 2.0.

Now that we've seen one major protocol built on top of OAuth 2.0, let's take a closer look at several more that are solving advanced use cases.